2

RL-TR-91-274, Vol Va (of five)
Final Technical Report
November 1991

PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Penelope User Guide: Larch/Ada Reference Manual

ORA Corporation

Sponsored by
Strategic Defense Initiative Office

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

PENELOPE: AN ADA VERIFICATION ENVIRONMENT,
Penelope User Guide: Larch/Ada Reference Manual

Carla Marceau

Contractor:	ORA Corporation
Contract Number:	F30602-86-C-0071
Effective Date of Contract:	19 Aug 86
Contract Expiration Date:	30 Sep 89
Short Title of Work:	PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Penelope User Guide: Larch/Ada Reference Manual
Period of Work Covered:	Aug 86 - Aug 89
Principal Investigator:	Maureen Stillman
Phone:	(607) 277-2020
RL Project Engineer:	John C. Faust
Phone:	(315) 330-3241

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by John C. Faust, RL/C3AB, Griffiss AFB 13441-5700, under contract F30602-86-C-0071.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE November 1991	3. REPORT TYPE AND DATES COVERED Final Aug 86 - Aug 89

4. TITLE AND SUBTITLE PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Penelope User Guide: Larch/Ada Reference Manual	5. FUNDING NUMBERS C - F30602-86-C-0071 PE - 35167G/63223C PR - 1070/B413 TA - 01/03 WU - 02
6. AUTHOR(S) Carla Marceau	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) ORA Corporation 301A Dates Drive Ithaca NY 14850-1313	8. PERFORMING ORGANIZATION REPORT NUMBER ORA TR 17-8
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Strategic Defense Initiative Office, Office of the Secretary of Defense Wash DC 20301-7100 Rome Laboratory (C3AB) Griffiss AFB NY 13441-5700	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-91-274, Vol Va (of five)

11. SUPPLEMENTARY NOTES
RL Project Engineer: John C. Faust/C3AB/(315) 330-3241

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This reference manual describes the specification language for the Penelope verification environment. It provides a brief, informal account of the syntax and semantics of Penelope annotations.

14. SUBJECT TERMS Ada, Larch, Larch/Ada, Formal Methods, Formal Specification, Program Verification, Predicate Transformers, Ada Verification	15. NUMBER OF PAGES 48
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Contents

Chapter 1

Introduction

This manual describes the syntax and static semantics of Larch/Ada, the specification language used in the Penelope verification editor for Ada. Earlier versions of the material contained in this document were combined with motivational material and some description of the semantics of Larch/Ada in *A Short Introduction to Larch/Ada-88*. That material is now found in [1]. Overviews of Penelope can be found in [8] and [6]. An introduction to using Penelope is provided in [4]. This manual is intended for the user who wishes to write specifications and develop programs using Penelope. It describes informally only the part of the language that has been implemented. It should not be read as a formal description of a full language for specifying Ada, which we refer to as Larch/Ada, and which is discussed in [1].

David Luckham's work with Anna [5], inspired the first version of Larch/Ada and in previous versions of this document Larch/Ada was called PolyAnna, in honor of that original inspriation. We have adopted the Larch approach to specification in choosing to separate the specification of theories from the specification of code [3].

1.1 Specification and proof in Larch/Ada

A Larch/Ada package or subprogram is *specified* by annotating its Ada code with subprogram and other annotations. These annotations contain assertions that are required to hold in designated program states. In specifications, the states of interest are typically the state on entry to a subprogram, and the states on normal or exceptional exit from a subprogram. The assertions are terms in first order logic. (The theory of the predefined Ada types, which defines the predefined operations of Ada, is described in [2].) Informally, a Larch/Ada package or subprogram *satisfies* its formal specification if all the assertions in the Larch/Ada annotations hold at appropriate control points.

The body of a Larch/Ada subprogram may be further *annotated* with annotations such as invariant annotations and embedded assertions.

Programmers develop annotated Larch/Ada programs using the Penelope editor, which, given the specification of a subprogram, generates preconditions during program development. (The preconditions are generated incrementally, which means that every time a programmer makes a change to a program, the preconditions immediately reflect the effects of that change.) The editor also generates *verification conditions* (usually one per loop plus one per subprogram body). The verification conditions are purely logical statements, the proof of which guarantees that the program satisfies its specification.The Penelope editor needs constant access to a theorem prover, both to help simplify the incremental preconditions and to prove the verification conditions. We have integrated a simple proof checker for first order logic into Penelope.

This document describes informally the Larch/Ada annotations and what it means for an annotated program to be correct.

1.2 Larch

As designers of programs to be verified, we often have difficulty writing down a formal specification that captures the meaning we intend for the program. The Larch Shared Language is designed to help us write readable formal specifications. The Larch Shared Language may be used in the specification of programs written in any language; it is the mathematical component of a specification language, and it is used to write complex specification concepts and theories in units called *traits.* A Larch Interface Language is needed to apply shared-language concepts to a particular program. Larch/Ada is a Larch Interface Language for Ada. We refer the reader to Guttag, Horning, and Wing [3] for a discussion of Larch.

The manual presents terms and assertions, a simple trait-like facility in Penelope, and annotations. Some simple examples are provided.

Chapter 2

Terms and Assertions

2.1 Terms

The Ada language is designed for computation, not for reasoning. We use Ada expressions to instruct a machine about *what computations to perform*. We use Larch/Ada terms to denote the possible results of such computations.

Terms are used to denote values underlying Ada objects. Such values include constants (such as `1`, `2` , `true`, etc.) and the result of applying operators to terms (e.g., `1+x`, `not P`, `min(x,y)`). Predefined Larch/Ada operators exist for boolean and integer values, as well as for array and record objects. Larch/Ada operators are total mathematical functions. Since they are not computationally defined, they do not execute, terminate or raise exceptions. They simply denote mathematical values.

Larch/Ada is a *sorted* language. Just as each expression in Ada has a type, so each term in Larch/Ada has a sort. Larch/Ada sorts may be roughly understood as the the mathematical domains underlying the Ada types. Larch/Ada terms are sorted. For example, `x + true` is not permitted. More information about the sorts and the semantics of terms may be found in [1].

The syntax of terms is:

$\langle term \rangle \rightarrow$
  `true`
  | `false`

| ⟨*integer*⟩
| ⟨*variable*⟩
| ⟨*unary operator*⟩ ⟨*term*⟩
| ⟨*term*⟩ ⟨*binary operator*⟩ ⟨*term*⟩
| ⟨*function application*⟩
| ⟨*conditional term*⟩
| ⟨*modified term*⟩
| ⟨*quantified term*⟩
| ⟨*array term*⟩
| ⟨*record term*⟩
| ⟨*skolem term*⟩

An *assertion* is a boolean term. Assertions are used in specifications, e.g. to represent input and output conditions for subprograms.

⟨*assertion*⟩ → `term`

2.2 Constants

The boolean values `true` and `false`, as well as the integers, are predefined in Larch/Ada.

2.3 Variables

⟨*variable*⟩ →
  ⟨*identifier*⟩
⟨*identifier*⟩ →
  any string of alphabetic/numeric or underscore characters
    beginning with an alphabetic character

The meaning of an identifier in a term depends on the context in which it appears. There are three possibilities. First, the identifier may be a bound variable occurring in a quantified term. For example, in the term

```
forall i,j:: b[i,j]=0
```

the variables `i` and `j` are bound. Second, at any point in an Ada program text the current *state* associates certain identifiers (those appearing in declarations in the current declarative region, for example) with Ada program objects. In the above example, b may be such an Ada variable. Third, the variable may be a free logical variable, as in

```
P or not P.
```

It may happen that there is more than one possible meaning for an identifier. In case of conflicts, a free variable refers to the Ada object of that name that is directly visible in the current state, if any. Otherwise it is a free logical variable.

Note that free logical variables are not permitted in annotations of an Ada program, although they may occur in traits and proofs.

2.4 Unary and binary operators

```
⟨unary operator⟩ →
  + | - | abs | not
⟨binary operator⟩ →
  and | or | xor | ->
  | = | /= | < | <= | > | >=
  | + | - | & | * | / | mod | rem | **
```

Larch/Ada operators are defined corresponding to most of Ada's unary and binary operators on integers. It is important to remember that they refer to total mathematical functions and never "fail" or raise exceptions. In cases where an Ada expression raises an exception, the corresponding Larch/Ada term may denote a value, e.g. x/0, for which there is no corresponding value in the Ada type. Examples of unary and binary operators in terms are a+b, x=6 or x>7. Note that the Ada short circuit control forms **and then** and **or else** do not correspond to any operators of the term language.

The following table summarizes the associativity and precedence of the supported predefined operators.

Operator	Associativity
AND, OR, XOR	left
<, <=, >, >=, =, /=	none
+, -, &	left
unary +, unary -,	none
*, /, MOD, REM	left
NOT, ABS, **	none

2.5 Function application

$\langle$*function application*$\rangle \rightarrow$
  $\langle$*designator*$\rangle$($\langle$*termlist*$\rangle$)
$\langle$*termlist*$\rangle \rightarrow$
  $\langle$*term*$\rangle$
 | $\langle$*term*$\rangle$, $\langle$*termlist*$\rangle$

The user may define mathematical functions and apply them to arguments. The designator never refers to an Ada *function*. Thus the same identifier may be used for an Ada function and a mathematical function without ambiguity. It is preferable, however, to choose distinct names for mathematical functions in order to avoid confusion for the human reader.

Predefined functions may also use the function application syntax.

2.6 Two-state terms

A *state* σ is a function that to every program object associates a value. We often use the terminology "the value of a variable (or object) in state σ." States are important because the effects of executing an Ada program can be described by describing the concomitant changes in the values of program objects, i.e. the changes in state.

The notion of state can be extended so that a state σ associates a value to every term. The value has the same sort as the term. If a term has free variables denoting program objects, the only way we can figure out what value that term denotes is to apply a state to it.

Three different states are of interest in the annotation of an Ada subprogram.

exit The subprogram annotation makes claims about the values of Ada objects on exit from the subprogram.

entry The subprogram annotation also makes assumptions about values of Ada objects on entry to the subprogram.

current Other annotations (embedded assertions, loop invariants, etc.) may make claims about the values of objects in the current state, i.e., the state at that point in the program.

It may happen that in an exit annotation we wish to refer to the value of a variable on exit from and also on entry to the subprogram, for example to say that the subprogram increments the entry value. The reserved word **in** designates the value of a variable or term in the entry state.

$\langle$*modified term*$\rangle \rightarrow$
  in $\langle$*variable*$\rangle$
| **in** $\langle$*term*$\rangle$

To specify a sort subprogram, for example, we might write:

```
type intarray is array (integer) of integer;
procedure sort_array (in out a: intarray);
--| where
--|     out (permutation(a, in a) and sorted(a));
--| end where;
```

where `permutation` and `sorted` come from the theory of arrays.

2.7 Conditional terms

$\langle$*term*$\rangle \rightarrow$
  `if` $\langle$*term*$\rangle$ `then` $\langle$*term*$\rangle$ `else` $\langle$*term*$\rangle$

The last two subterms must be of the same sort, and the first subterm must be boolean. If Q and R are boolean terms, then the term `if P then Q else R` is equivalent to $(P \wedge Q) \vee (\neg P \wedge R)$.

The following example shows a boolean conditional term and also an integer conditional term.

```
function abs_max (a,b: in integer) return integer;
--| where
--|     in if a>=0 then b>=0 else b<0;
--|     return if a>0 then max(a,b) else max(-a,-b);
--| end where;
```

2.8 Quantified terms

⟨*term*⟩ →
  `forall`⟨*idlist*⟩ `::`⟨*term*⟩
 | `exists` ⟨*idlist*⟩`::` ⟨*term*⟩
⟨*idlist*⟩ →
  ⟨*identifier*⟩ | ⟨*identifier*⟩ ⟨*idlist*⟩

The subterm of a quantified term must be boolean. Example:

```
type intarray is array (integer) of integer;
function array_max (a: in intarray; n: in integer) return integer;
--| where
--|     return z such that forall i::i>0 and i<n -> a[z]>=a[i];
--| end where;
```

2.9 Array terms

⟨*array terms*⟩ →
  ⟨*term*⟩`[`⟨*termlist*⟩`]`
 | ⟨*term*⟩`[`⟨*termlist*⟩`=>`⟨*term*⟩`]`

If a is an array then `a[i,j]` represents the value of a component of a. It is often useful to represent the value of a if an component is replaced. Suppose we replace component `i` of a with v. We can represent the resulting value by `a[i => v]`.

2.10 Record terms

> ⟨*record terms*⟩ →
>   ⟨*term*⟩`.`⟨*termlist*⟩
> | ⟨*term*⟩`[.`⟨*termlist*⟩`=>`⟨*term*⟩`]`

If `r` is a record then `r.f` represents field f of `r`. It is often useful to represent the value of `r` if an component is replaced. Suppose we replace component `f` of `r` with `v`. We can represent the resulting value by `r[.f=>v]`.

2.11 Terms that may be produced by the editor

There are some terms that are not entered by the user, but may be produced by the editor. These are hidden Ada variables and skolem functions. A ⟨*hidden Ada variable*⟩ is used to represent an Ada variable at some point in the program where that variable is not visible, e. g., because it has been hidden by a declaration in an interior declarative region. A ⟨*hidden Ada variable*⟩ associates a context (declarative region) with a variable name. Contexts are usually very long and hard to read. Such names can be eliminated, if desired, by avoiding duplicate names.

A skolem function supplies a name for a value returned by a subprogram. Skolem functions are long and hard to read. They have the form

> ⟨*skolem function*⟩ →
>   `Func<`⟨*name for Ada function*⟩`,`⟨*variable*⟩`>(`⟨*termlist*⟩`)`

Skolem functions usually occur when a subprogram returns a value and its specification does not explicitly state the value returned. For example,

```
function return_one (a: in integer) return integer;
--| where
--|     return z such that z>0 and z<2;
--| end where;
```

Better:

```
function return_one (a: in integer) return integer;
--| where
--|     return 1;
--| end where;
```

Chapter 3

A Simplified Trait Facility for Penelope

In the Larch two-tiered approach to specification, a programmer would naturally develop in the Larch Shared Language a body of mathematics that he would then appeal to in Larch/Ada or other Larch interface language specifications. Implementation of such an approach implies some way of appealing from the program specification to the trait, for example a library of traits. In the absence of any such facility, Penelope provides a very much simplified trait facility. It is not intended to replace a proper facility for building traits in the Larch Shared Language, but merely to allow the user to enter into the editor some information contained in such traits so that he can appeal to it in specification and verification of his programs.

3.1 Traits

In Penelope the user can enter information from traits above the program text. The scope of trait information is the entire program. Multiple traits may be entered. The axioms and lemmas available are simply the union of all the axioms and lemmas entered in the traits. No checks are provided for consistency, nor is the user required to show that the lemmas follow from the axioms. In the future the user will be able to enter Larch traits and the Larch checker will be used to check the traits for correctness.

```
⟨trait⟩ →
  --| trait ⟨identifier⟩ is
  ⟨function definitions⟩
  --| axioms: ⟨labelled_assertions⟩
  --| end axioms;
  --| lemmas: ⟨labelled_assertions⟩
  --| end lemmas;
```

3.2 Function signatures

```
⟨function definition⟩ →
  --| introduces ⟨identifier⟩: ⟨signature⟩;
⟨signature⟩ →
  [⟨idlist⟩]->⟨identifier⟩
```

Note that a constant is a nullary function. Currently two functions may not have the same name. In this future this restriction will be relaxed so that two functions may have the same name if their *signatures are distinct*. The user may not redefine predefined symbols (such as "+" on integers).

3.3 Axioms and Lemmas

Axioms and lemmas are assertions. They may contain free variables but may not reference Ada variables. They may use the predefined mathematics for Ada records, arrays, etc. In proofs and in simplification directives the axioms and lemmas are referred to by their labels.

```
⟨labelled_assertion⟩ →
  ⟨identifier⟩:⟨assertion⟩;
```

Chapter 4

Annotations

4.1 Subprogram annotations

A *subprogram annotation* represents a contract between the subprogram and its callers. The subprogram annotation states what must be true when the subprogram is called (the responsibility of the caller) and what is then guaranteed to be true if the subprogram terminates. Externally, every caller must show that the input conditions of the subprogram are satisfied at the point of the call, and may assume that the exit conditions hold if the subprogram returns. Internally, the implementation of the subprogram must be such that if the input conditions hold and the subprogram terminates, then the exit conditions can be proved to hold.

Note that in the above discussion we do not assume that the subprogram must terminate. That is, subprogram annotation specifies conditions for the *partial correctness* of the subprogram, as opposed to *total correctness*, which additionally specifies that the program must terminate.

Entry state, exit state, and two-state predicates Recall that in a *two-state predicate* subterms of the assertion may be modified by in, and such subterms get their values from the entry state. Other subterms get their values from the current or exit state. In a subprogram annotation using a two-state predicate, the entry state is the state on entry to the subprogram, and the exit state is the state on termination (which may be normal or exceptional according to the annotation).

Assertions embedded in subprograms are also two-state predicates: unmodified terms get their values from the current state, while terms modified by **in** get their values from the state on entry to the subprogram.

Syntax of subprogram annotations Subprogram annotations may follow subprogram declarations, or may precede the reserved word **is** in a subprogram body. Thus, in Larch/Ada:

```
⟨subprogram declaration⟩ →
   procedure ⟨identifier⟩ [⟨formal part⟩] ;
   ⟨subprogram annotation⟩
 | function ⟨designator⟩ [⟨formal part⟩] return ⟨type mark⟩;
   ⟨subprogram annotation⟩

⟨subprogram body⟩ →
   procedure ⟨identifier⟩ [⟨formal part⟩]
   ⟨subprogram annotation⟩
   is ⟨body⟩
 | function ⟨designator⟩ [⟨formal part⟩] return ⟨type mark⟩
   ⟨subprogram annotation⟩
   is ⟨body⟩
```

The syntax of the ⟨*subprogram annotation*⟩ is:

```
⟨subprogram annotation⟩ →
     --| where
     [⟨side effect annotations⟩]
     [⟨in annotations⟩]
     [⟨out annotations⟩]
     [⟨result annotations⟩]
     [⟨propagation constraints⟩]
     [⟨propagation promises⟩]
     --| end where;
```

4.1.1 Side effect annotations

Syntax:

⟨*side effect annotation*⟩ → `--|` `global` ⟨*formal part*⟩;

The ⟨*formal part*⟩ lists the global objects read and written by this subprogram. For example, suppose we wish to implement a stack package for a particular stack, called `my_stack`. Mathematics provides us with functions `top` and `pop` that respectively pick off the top element of the stack and return the rest of the stack. We may wish to implement a `pop_stack` function in which the top element is removed from the stack and returned to the caller. Thus there is a side effect on `my_stack`.

```
function pop_stack () return integer;
--| where
--|      global my_stack: in out;
--|      out my_stack=mathematical_pop(in my_stack);
--|      return mathematical_top(in my_stack);
--| end where;
```

Note that the parameter names appearing in the formal part are the names of visible global objects. They are called the *global parameters* or sometimes the *implicit parameters* of the subprogram.

A global variable may ocur at most once in the side effect annotations for a program.

For any fragment of an Ada program, we can determine statically what global objects are read and written by that fragment. The side effect annotation of a subprogram must list all objects read or written by the program.

4.1.2 In annotations

Syntax:

⟨*in annotation*⟩ → `--|` in ⟨*assertion*⟩;

where the assertion is not a two-state predicate. The only Ada variables allowed to appear in the assertion are the global or formal parameters of modes `in` or in out.[1] If the in annotation is omitted, that is equivalent to an in annotation with an ⟨*assertion*⟩ of *true*.

The implementor is allowed to assume that, on entry to the subprogram, the state satisfies the assertion. Users of the subprogram must show that the state immediately preceding the call satisfies the assertion (when the values of the appropriate actual parameters are substituted for the formal parameters).

4.1.3 Out annotations

Syntax:

⟨*Out annotation*⟩ → `--| out` ⟨*assertion*⟩`;`

where the assertion is a two-state predicate. Unless preceded by the modifier in, all variables get their values from the exit state. The only Ada variables allowed to appear in the assertion are those appearing in the formal parts of the subprogram declaration and the subprogram's side effect annotations. If the out annotation is omitted, that is equivalent to an out annotation with an ⟨*assertion*⟩ of *true*.

Verification conditions for the subprogram are generated whose truth will guarantee that, if the subprogram is called in a state satisfying the in annotation, and if it terminates normally (i.e. without propagating an exception) , the state after termination will satisfy the out annotation.

The out annotation can be used to annotate both procedures and functions, although one must use a result annotation to be able to refer to the value returned by a function.

4.1.4 Result annotations

Syntax:

⟨*result annotation*⟩ →
    `--| return` ⟨*identifier*⟩ `such that` ⟨*assertion*⟩`;`

[1]This is a simplification. Some attributes of formal parameters of mode out may be known on entry, for example A'FIRST when A is an array.

where the assertion is a two-state predicate. The only Ada variables allowed to appear in the assertion are those variables appearing in the formal parts of the subprogram declaration and the subprogram's side effect annotation, and ⟨*identifier*⟩. The ⟨*identifier*⟩ may not appear in the ⟨*assertion*⟩ modified by in, since it is senseless to talk about the value on entry of the thing returned.

The result annotation may annotate only functions. It is exactly like the out annotation except that the ⟨*identifier*⟩ stands for the return value. In principle the result annotation renders the out annotation superfluous for functions, but the out annotation may be clearer to a reader in cases where it can be used. If the result annotation is omitted, that is equivalent to a result annotation with an ⟨*assertion*⟩ of *true*.

We offer a short form result annotation

> ⟨*result annotation*⟩ → `--| return` ⟨*term*⟩`;`

which is equivalent to

> ⟨*result annotation*⟩ →
>   `--| return` ⟨*name*⟩ `such that` ⟨*name*⟩ `=` ⟨*term*⟩`;`

where ⟨*name*⟩ is an identifier that is not free in ⟨*term*⟩. Thus the annotation

```
--| return x*y;
```

is equivalent to

```
--| return z such that z = x*y;
```

4.1.5 Propagation constraints

Syntax:

> ⟨*propagation constraint*⟩ →
>   ⟨*constraint propagation annotation*⟩
> | ⟨*strong propagation annotation*⟩
> | ⟨*exact propagation annotation*⟩

Constraint propagation annotation Syntax:

> ⟨*constraint propagation annotation*⟩ →
> `--| raise` ⟨*exception*⟩ [| ⟨*exception*⟩...] `=>` in ⟨*assertion*⟩;

where the ⟨*assertion*⟩ is not a two-state predicate. All Ada variables in the assertion take their values from the entry state. The only Ada variables allowed to appear in the assertion are the global or formal parameters of modes `in` or `in out`.

If the subprogram terminates by propagating any of the exceptions listed, the entry state must have satisfied ⟨*assertion*⟩. Verification conditions will be generated whose truth will guarantee that the subprogram cannot propagate any of the exceptions listed unless it is called in a state satisfying ⟨*assertion*⟩.

Strong propagation annotation Syntax:

> ⟨*strong propagation annotation*⟩ →
> `--| in` ⟨*assertion*⟩ `=> raise` ⟨*exception*⟩ [| ⟨*exception*⟩...];

where the ⟨*assertion*⟩ is not a two-state predicate. All Ada variables in the assertion take their values from the entry state. The only Ada variables allowed to appear in the assertion are the global or formal parameters of modes `in` or `in out`.[2]

When the entry state satisfies ⟨*assertion*⟩, the subprogram *must* raise one of the exceptions listed, if it terminates.[3] Therefore, strong propagation annotations for disjoint sets of exceptions must have mutually exclusive assertions in order for the program to be proved correct. Verification conditions will be generated whose truth will guarantee this exclusivity, and will guarantee that every time it is called in a state satisfying ⟨*assertion*⟩ it will propagate one of the exceptions listed if it terminates at all.

[2]In the propagation annotations, in acts as a syntactic marker, not as a modifier.

[3]This is subject to the assumption that the subprogram does not terminate by propagating `storage_error` or `numeric_error`, and that no undetected numeric overflow occurs during execution of the subprogram.

Exact propagation annotations

> ⟨*exact propagation annotation*⟩ →
> `--|` **raise** ⟨*exception*⟩ [`|` ⟨*exception*⟩...] `<=>` **in** ⟨*assertion*⟩;

or

> ⟨*exact propagation annotation*⟩ →
> `--|` **in** ⟨*assertion*⟩ `<=>` **raise** ⟨*exception*⟩ [`|` ⟨*exception*⟩...];

where the ⟨*assertion*⟩ is not a two-state predicate. All Ada variables in the assertion take their values from the entry state. The only Ada variables allowed to appear in the assertion are the global or formal parameters of modes `in` or `in out`.

This annotation is an abbreviation for both the strong propagation annotation and the constraint propagation annotation with the same list of exceptions and ⟨*assertion*⟩. The same interpretations and restrictions apply; the intent is that ⟨*assertion*⟩ be a necessary and sufficient assertion for the propagation by the subprogram of one of the exceptions listed, if the program terminates.

4.1.6 Propagation promises

Syntax:

> ⟨*propagation promise*⟩ →
> `--|` **raise** ⟨*exception*⟩ [`|` ⟨*exception*⟩...] [`=>` **promise** ⟨*assertion*⟩];

where the ⟨*assertion*⟩ is a two-state predicate. Unmodified variables take their values from the exit state. If the `promise` clause is omitted that is equivalent to a `promise` clause with an ⟨*assertion*⟩ of *true*. (This only makes sense in the case of a subprogram that can raise an exception with completely unspecified results.) If the subprogram terminates by propagating any of the exceptions listed, it does so in a state satisfying the ⟨*assertion*⟩.

The following kinds of Ada variables may appear in the ⟨*assertion*⟩:

- global parameters
- formal parameters of mode `in`

- formal parameters of mode in out, but only modified by in

We cannot make general statements about formal parameters of mode in out, because we mustn't be able to distinguish methods of parameter passing by looking at the values of in out parameters after exceptional termination. (There is no difficulty with global parameters, since they are "passed by name" always.) Verification conditions will be generated whose truth will guarantee that the exit state satisfies ⟨*assertion*⟩ whenever the subprogram terminates by propagating any of the exceptions named.

4.1.7 Summary of propagation annotations

(In this table we use ⟨*assn*⟩ for ⟨*assertion*⟩ in order to save space.)

Annotation	**Description**
`--| raise ⟨list⟩ => promise ⟨assn⟩;`	Whenever an exception on the list is propagated, the assertion holds in the propagating state. The assertion may hold in the exit state even if none of the exceptions listed is propagated.
`--| raise ⟨list⟩ => in ⟨assn⟩;`	Whenever an exception on the list is propagated, the assertion held in the initial (calling) state. The assertion may have held in the initial state even if none of the exceptions listed is propagated.
`--| in ⟨assn⟩ => raise ⟨list⟩;`	Whenever the assertion held in the initial state, the subprogram terminates by propagating one of the exceptions on the list, if it terminates at all. The subprogram may terminate by propagating one of the exceptions listed even if the assertion did not hold in the initial state.
`--| in ⟨assn⟩ <=> raise ⟨list⟩;`	The subprogram terminates by propagating one of the exceptions on the list *if and only if* it terminates and the assertion held in the initial state. The subprogram may *not* terminate by propagating one of the exceptions listed if the assertion did not hold in the initial state, and the assertion must *not* have held in the initial state if the subprogram terminates and none of the exceptions listed is propagated.

4.2 Embedded assertions

The user may strengthen the claims made in a subprogram annotation by using an *embedded assertion.* Syntactically, an embedded assertion is a formal comment, thus:

⟨*embedded assertion*⟩ → `--|` ⟨*assertion*⟩`;`

where the syntax of ⟨*assertion*⟩ is outlined in Chapter 2. The embedded assertion may appear only in the position of a declaration in a declarative part, or in the position of a statement in a sequence of statements. Such a location is called a *control point.* Verification conditions are generated for the program in which the assertion is embedded. The truth of the verification conditions will guarantee that, whenever control reaches an embedded assertion, the program state will satisfy that assertion. The methods of VC generation are given by Wolfgang Polak [7]. The embedded assertion is a two-state predicate (see above, Section 2.6).

4.3 Cut point assertions

A *cut point assertion* is similar to an embedded assertion, but whereas an embedded assertion makes a partial claim about the current state (the assertion is true in this state), the cut point assertion makes a more total claim. It says that the truth of the ⟨*assertion*⟩ at this point follows from the input conditions of the subprogram and is sufficient to show the exit conditions of the program. Syntactically, a cut point assertion is a formal comment, thus:

⟨*cut point assertion*⟩ → `--|assert` ⟨*assertion*⟩`;`

where the syntax of ⟨*assertion*⟩ is outlined in Chapter 2. Cut point assertions may appear where embedded assertions may appear. A verification condition is generated for each cut point assertion; its truth will guarantee that that if the ⟨*assertion*⟩ holds whenever control reaches that point, then the exit conditions of the program will be satified when the program terminates. The truth of the verification conditions generated for the subprogram will guarantee that whenever control reaches the cut point assertion the program state will satisfy the ⟨*assertion*⟩.

4.4 Annotations of loops

Loop invariants Each loop in an Ada program requires an invariant that summarizes the content of the loop. A verification generation is generated to guarantee that the invariant is preserved by the loop body. The user may provide an invariant explicitly using the `invariant` keyword.

```
--| invariant ⟨assertion⟩;
```

In Larch/Ada the user can choose whether or not to distinguish a special assertion as a loop invariant. If the user decides not to provide a loop invariant, the VC generation procedure can synthesize a loop invariant from assertions embedded in the loop.

Chapter 5

Very Simple Examples

5.1 A simple example involving a loop

The following example is a simple subprogram that does multiplication by repeated addition.

```
procedure mult (x,y : in integer; z: out integer)
  --| where
  --|   in (x >= 0);
  --|   out (z=(x * y));
  --| end where
is
  u: integer := x;
begin
  z := 0;
  iter: loop
    --| invariant ((x * y)= (z + (u * y)));
    exit iter when (u = 0);
    u := (u - 1);
    z := (z + y);
  end loop;
end mult;
```

5.2 Exceptions

We can modify the above example slightly if the subprogram is to terminate exceptionally when given the "wrong" input data. We use an exact propagation constraint.

```
procedure mult(x,y: in integer; z: out integer)
  --| where
  --|   out (z=(x * y));
  --|   raise input_error <=> (x < 0);
  --| end where
is
  u: integer := x;
begin
  if (x < 0) then
    raise input_error;
  end if;
  z := 0;
  iter: loop
    --| invariant ((x * y)= (z + (u * y)));
    exit iter when (u = 0);
    u := (u - 1);
    z := (z + y);
  end loop;
end mult;
```

Appendix 1

Subset of Ada supported

This appendix informally describes the subset of Ada supported by Penelope at the time of publication. Many of the features not yet supported by the software are supported by the theory [7].

1.1 Lexical elements

Identifiers, decimal integer literals and comments are supported. Reals, based literals, string and character literals and pragmas are not supported.

1.2 Data types

Supported:

- predefined integer and boolean types
- enumeration types
- record types (without variants)
- (constrained) array types

Not supported:

- subtypes

- access types
- derived types
- renaming declarations
- number declarations

1.3 Operations and expressions

Supported:

- logical, relational and arithmetic operators on integers

Not supported:

- array slices
- attributes
- aggregates
- short-circuit control forms
- type conversions
- qualified expressions

1.4 Statements

Supported:

- assignment
- if statements
- loops (except `for` loops)
- block
- exit

- return

Not supported:

- array assignment
- go to

1.5 Subprograms

Supported:

- procedures and functions. Note: there is a conservative requirement to avoid aliasing. Arguments to subprograms (including implicit or global arguments) must be pairwise independent. That is, let *reads*(E) be the program objects potentially read during evaluation of E and *writes*(E) be the program objects potentially written during evaluation of E. Then E_1 and E_2 are independent if and only if
$$\begin{aligned} reads(E_1) \cap writes(E_2) &= \emptyset \wedge \\ writes(E_1) \cap reads(E_2) &= \emptyset \wedge \\ writes(E_1) \cap writes(E_2) &= \emptyset \end{aligned}$$
Thus if *swap* is a function with two inout parameters and a is an array, then `swap(a(i),a(j))` is not allowed.
- overloading of operators and subprogram names

Not supported:

- default parameters

1.6 Packages

Supported:

- packages

Not supported:

- private and limited types
- deferred constants

1.7 Exceptions

User-defined exceptions are supported.

1.8 Other

Representation- and implementation-dependent features are not supported. Tasking, generics and input-output are not supported.

Bibliography

[1] Odyssey Research Associates. The Larch/Ada rationale. Technical Report TR-17-8, Odyssey Research Associates, 1989.

[2] David Guaspari. Introduction to domains for Ada types. Technical Report 17-2, Odyssey Research Associates, 1989.

[3] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report TR 5, DEC/SRC, July 1985.

[4] C. Douglas Harper. Guide to the Penelope editor. Technical Report TR-17-9, Odyssey Research Associates, 1989.

[5] D. C. Luckham et al. Anna: A language for annotating Ada programs. Technical Report CSL-84-261, Stanford University, 1986. Reference Manual.

[6] Carla Marceau and C. Douglas Harper. An interactive approach to Ada verification. In *COMPASS89 Proceedings*, June 1989.

[7] Wolfgang Polak. Predicate transformer semantics for Ada. Odyssey Research Associates internal document, 1988.

[8] Norman Ramsey. Developing formally verified Ada programs. In *Proceedings of the Fifth International Conference on Software Specification and Design*, May 1989.

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.